Part VII

Integrated Environments

20 Integrated development using ECLIPSE

In the previous chapters we presented a number of tools that are helpful when working on software projects. However, these tools differ from each other in the operation and in the presentation of the outcomes. *Integrated environments* try to combine these tools with each other under the same graphical interface and make them intuitively operational. The following requirements are made on integrated environments:

Integrated environments

Intuitive operation. It should be possible to use it without a handbook or any training.

Automation. Routine tasks should be carried out as automatically as possible or by using short commands (a keystroke).

Interoperability. The outcomes and data of a tool should be able to be used by other tools without any trouble.

Abstraction. The environment should support as many programming languages, operating systems, and kinds of tools as possible in order to be considered abstractly from their special features.

Functionality. It should support as many phases of the software development as possible.

Most commercial compilers come in an *integrated development environment* (*IDE*). However they provide more than support in the *edit-compile-debug cycle*. Furthermore, IDEs are mostly connected to a specific programming language, a specific compiler and debugger, and a specific operating system; they do not provide any way to integrate external tools—therefore they are referred to as a *closed* environment. On the other hand, an *open* environment provides a way to integrate other tools, as well as interfaces for accessing data and functions of the environment. An environment of this kind is ECLIPSE, which we will introduce in this chapter.

Integrated development environment, IDE

Open and closed environments

ECLIPSE has been developed by Object Technology International (OTI) and was donated as open source by IBM. During the e-hype the name

ECLIPSE has been chosen to express that ECLIPSE outshines existing IDEs. However, in order to invite SUN to join ECLIPSE and avoid a renaming, the official source for the name is the Pink Floyd album “Dark Side Of The Moon” in which a song is called “Eclipse”.

plug-ins

ECLIPSE refers to itself as “an IDE for anything, and for nothing in particular”. The design of ECLIPSE reflects this principle: the core of ECLIPSE is small and compact, and the actual functionality is realized in *plug-ins*. By the use of the plug-in concept ECLIPSE is extensible and allows a seamless integration of third party tools into the ECLIPSE user interface.

Although ECLIPSE is implemented in JAVA, it does not run on all operating systems that are supported by JAVA. The reason is the *Standard Widget Toolkit* (SWT) which is only available for WINDOWS, MacOS, LINUX and some UNIX dialects. SWT is faster than the JAVA widgets and closer to the *look and feel* of the operating system. ECLIPSE itself only supports the development for the JAVA programming language. However, additional third-party plug-ins extend ECLIPSE for further programming languages, like C, C++, PYTHON, PERL und PHP.

ECLIPSE can be used in all phases of software development, especially in phases that deal with source code. These are the later phases of software development, like implementation and maintenance. ECLIPSE provides tool support for most activities that were introduced in previous chapters: version management (CVS), program building (ANT), software testing (JUNIT), and debugging. Furthermore, ECLIPSE provides various ways of visualizing source code properties and supports the refactoring of existing source code.

20.1 Projects in ECLIPSE

Projects

Software products consist of thousands of source files and dozens of libraries; therefore, they need to be divided into manageable parts. Most IDEs offer *projects* for this kind of structuring. Each project has a *project directory*, which contains the files belonging to the project. Besides source code, make files, and documentation, this directory also contains a *project description*. The project description consists of specific information about the project, such as the version management used, or dependencies to other projects.

Workspace

Projects in ECLIPSE

In ECLIPSE, each user has a *workspace* which contains one or more projects. For instance, Figure 20.1 on the facing page shows the workspace for ASPECTJ. The ASPECTJ tool consists of several modules, which are realized as projects in ECLIPSE. For each project, there are directories for the source files (`src`) and the binary files (`bin`). Furthermore, each project has a file called `.project` with project settings and in some cases a file called `.classpath` that contains the JAVA classpath. An additional project called

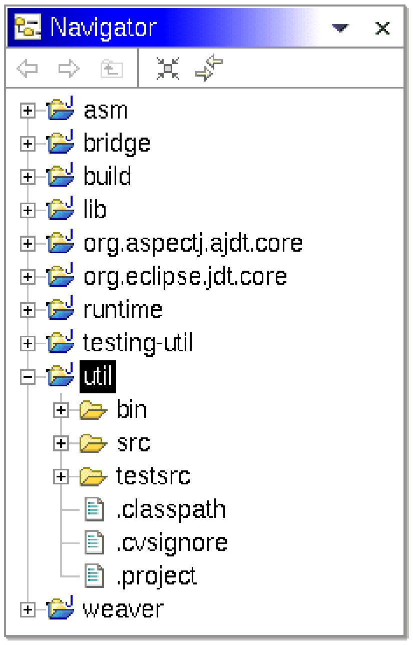


Figure 20.1
A sample workspace

`lib` collects all libraries used by ASPECTJ, and is included by the other projects.

The term *resources* summarizes projects, directories, and files in ECLIPSE. Resources can be tagged with *markers*. Internally, ECLIPSE uses such markers to highlight errors, warnings, or bookmarks in the source code.

Resources
Markers

20.2 The user interface

The graphical user interface of ECLIPSE is called the *workbench*. Besides the mandatory menu bars, tool bars, and status bars, the ECLIPSE workbench consists of *editors*, *views*, and *perspectives*.

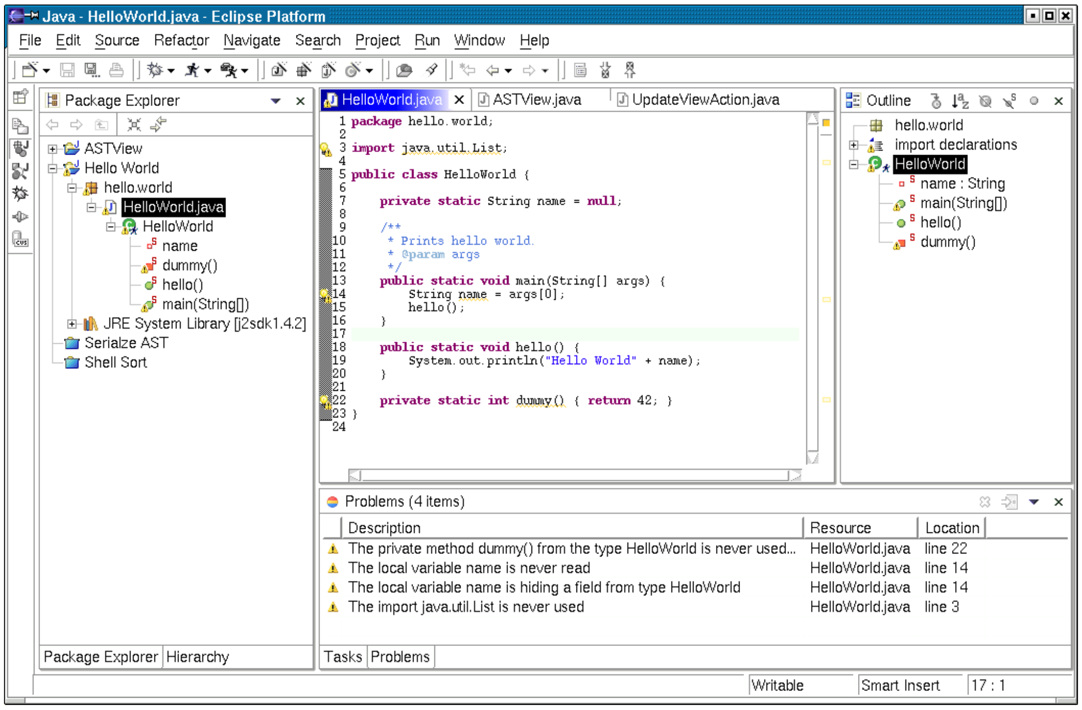
Workbench

20.2.1 Editors

With an *editor* a developer opens and changes files. An editor follows the *open-modify-save-cycle*. This means, only opened files can be modified, and changes take not effect until they are saved. Editors usually extend the ECLIPSE menu bars and tool bars with additional entries.

Editors

In Figure 20.2 three editors are opened: one for `HelloWorld.java`, `ASTView.java`, and `UpdateViewAction.java`, respectively. However, only one is visible because multiple editors are displayed in a *stacked* manner.



```
 1 package hello.world;
 2
 3 import java.util.List;
 4
 5 public class HelloWorld {
 6
 7     private static String name = null;
 8
 9     /**
10      * Prints hello world.
11      * @param args
12      */
13     public static void main(String[] args) {
14         String name = args[0];
15         hello();
16     }
17
18     public static void hello() {
19         System.out.println("Hello World" + name);
20     }
21
22     private static int dummy() { return 42; }
23 }
24
```

Problems (4 items)

Description	Resource	Location
The private method dummy() from the type HelloWorld is never used...	HelloWorld.java	line 22
The local variable name is never read	HelloWorld.java	line 14
The local variable name is hiding a field from type HelloWorld	HelloWorld.java	line 14
The import java.util.List is never used	HelloWorld.java	line 3

Figure 20.2
The ECLIPSE workbench

Views

20.2.2 Views

In contrast to editors, *views* focus on the presentation of information about an item. It is possible to change the item, but the changes take effect immediately. ECLIPSE provides many predefined views:

❏ The *Bookmarks View* contains all bookmarks.

❏ The *Navigator View* simplifies navigation through the workspace.

❏ The *Outline View* shows the structure of files.

❏ The *Problems View* displays warnings and errors.

❏ The *Properties View* contains properties of files, e.g., the size.

❏ The *Search View* lists results of a search.

❏ The *Tasks View* collects tasks for the user.

In Figure 20.2 several views are opened: the Outline View, the Tasks View (concealed), and the Problems View. Furthermore, two JAVA views are displayed: the *Package Explorer* and the *Hierarchy View* (concealed). The

Package Explorer simplifies—like the Navigator View—the navigation, but its main focus is on JAVA projects. The Hierarchy View visualizes inheritance hierarchies and will be discussed in detail in Section 20.5.1.

20.2.3 Perspectives

Perspectives

The selection and arrangement of editors and views depends heavily on the task of the user. For instance, during development she needs other views than during debugging. In order to enable a fast shift between tasks, ECLIPSE groups editors and views to *perspectives*. A perspective controls the arrangement and visibility of editors and views (*task orientation*). Furthermore, perspectives can restrict the visibility of resources and actions (*information filtering*). Several perspectives can be opened at a particular time, but only one perspective is visible. In practice, perspectives are defined and customized by the user. However, ECLIPSE provides many predefined perspectives:

Task orientation and information filtering

- for exploring the workspace: *Resource*
- for working with CVS: *Repository Exploring*, *Team Synchronizing*
- for searching errors: *Debug*
- for JAVA programming: *Java*, *Java Browsing*, *Java Type Hierarchy*
- for developing ECLIPSE plug-ins: *Plug-in Development*

In Figure 20.2 the JAVA perspective is opened. By the use of the horizontal tool bar on the top right, we can shift to other perspectives very fast, like to the Debug or CVS perspective.

20.3 Working in teams

In practice, a team works on several locations simultaneously, and each developer has an own workspace that needs to be synchronized with the other ones (for concepts of version control, see Chapter 3 and 4). ECLIPSE supports teamwork by a special team environment that simplifies management, sharing, and synchronization of resources. The version archive is called *repository* by ECLIPSE and is accessed with plug-ins called *repository providers*. ECLIPSE supports optimistic as well as pessimistic cooperation strategies.

Repository provider

In order to prevent an accidental loss of resources, ECLIPSE maintains a built-in *local history*. This history stores complete copies and not only differences like RCS. Besides recovery, the history is valuable for comparing two different versions of a file. The history is limited by maximum size and maximum age thresholds.

Local history

ECLIPSE supports CVS by default. However, other SCM products can be integrated with additional plug-ins. Such plug-ins exists for CLEARCASE, PERFORCE, PVCS, SOURCE INTEGRITY, SUBVERSION, VISUAL SOURCE SAFE, and many other.

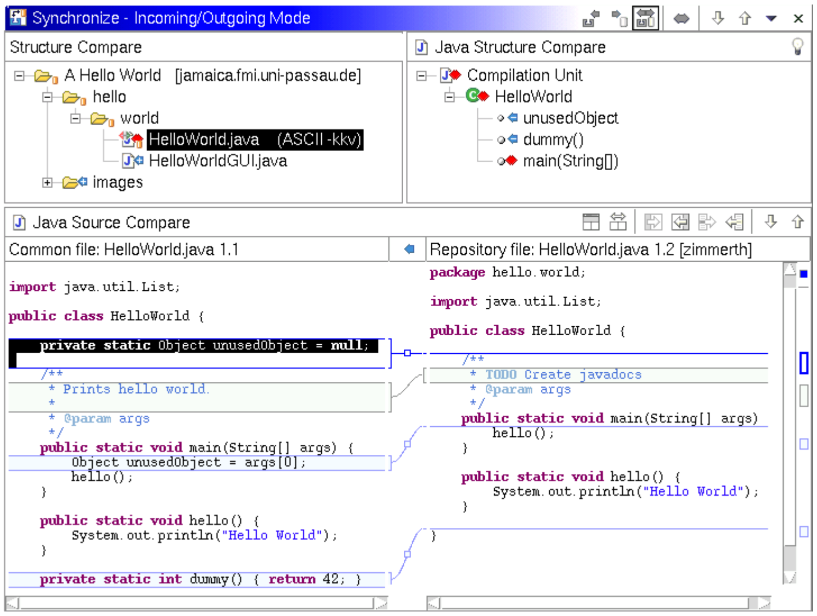
Integration of a repository provider

Although the functionality is the same for all SCM systems, the workflow is always different. In order to allow developers an easy use of familiar tools in ECLIPSE, repository providers have very much freedom in their realization. The integration in ECLIPSE takes place on two levels:

Hooks

Integration into the workspace. In some cases, a repository provider has to intervene or even prevent actions that change resources, e.g., modification of a locked file. For this purpose, ECLIPSE provides two *hooks*: Before moving or deleting a resource, ECLIPSE calls the *resource move/delete hook*, and before a file is opened or saved, it calls the *file modification validator*.

Integration into the workbench. The integration into the user interface is also passively. ECLIPSE defines placeholders for actions, preferences, and properties, but it is up to the repository provider to define concrete UI elements.


Synchronize - Incoming/Outgoing Mode
Structure Compare
A Hello World [jamaica.fmi.uni-passau.de]
hello
world
HelloWorld.java (ASCII -kkv)
HelloWorldGUI.java
images
Java Structure Compare
Compilation Unit
HelloWorld
unusedObject
dummy()
main(String[])
Java Source Compare
Common file: HelloWorld.java 1.1
Repository file: HelloWorld.java 1.2 [zimmerth]

```
import java.util.List;

public class HelloWorld {

    private static Object unusedObject = null;

    /**
     * Prints hello world.
     *
     * @param args
     */
    public static void main(String[] args) {
        Object unusedObject = args[0];
        hello();
    }

    public static void hello() {
        System.out.println("Hello World");
    }

    private static int dummy() { return 42; }
```

```
package hello.world;

import java.util.List;

public class HelloWorld {

    /**
     * TODO Create javadocs
     * @param args
     */
    public static void main(String[] args)
        hello();
    }

    public static void hello() {
        System.out.println("Hello World");
    }

}
```

Figure 20.3 The Synchronize View

Synchronize View

In ECLIPSE, the central view for version control with CVS is the *Synchronize View* (Figure 20.3), which is opened from the context menu by *Team* → *Synchronize with Repository*. This view compares the workspace with the

Incoming and outgoing changes

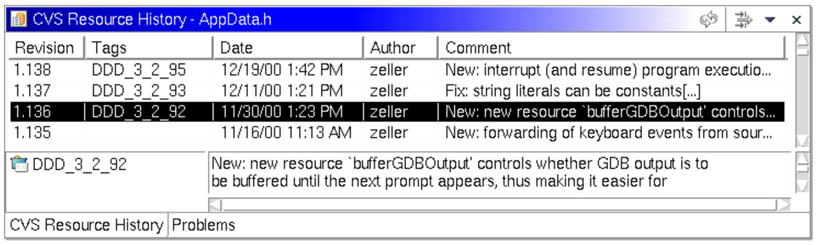
repository, and shows differences that are classified in *incoming* and *outgoing changes*. For incoming changes, the file in the CVS archive has been modified, and for outgoing changes the file in the workspace. Files that need to be integrated have incoming and outgoing changes, and are emphasized—like in Figure 20.3 the file `HelloWorld.java`. The directory `images` and the file `HelloWorldGUI.java` have been added to the repository and are incoming changes. ECLIPSE can sort changes by *change sets* that are changes grouped logically by comment, author, and date.

Change sets

If changes need to be integrated, ECLIPSE provides besides automatic integration a graphical *diff/merge* tool. Like SDIFF (Section 2.4), this tool compares both files vis-a-vis in two columns. Differences are highlighted and can be accessed easily. During integration, the developer decides for every difference, whether to use the version of the first or second file.

Structure-based comparison

In contrast to the line-based comparison of DIFF (Chapter 2), this tool compares hierarchical structures (*structure-based comparison*). In Figure 20.3 on the preceding page the structure of the file `HelloWorld.java` is compared. The tool recognizes that the method `dummy()` has been deleted. Furthermore it finds a conflict in the method `main` because it has been changed in both versions. On source code level of `main`, we notice that a local variable called `unusedObject` has been removed and the comment of `main` has been adjusted. The compare tool works on arbitrary files, even on the local history.



CVS Resource History - AppData.h

Revision	Tags	Date	Author	Comment
1.138	DDD_3_2_95	12/19/00 1:42 PM	zeller	New: interrupt (and resume) program executio...
1.137	DDD_3_2_93	12/11/00 1:21 PM	zeller	Fix: string literals can be constants[...]
1.136	DDD_3_2_92	11/30/00 1:23 PM	zeller	New: new resource `bufferGDBOutput' controls...
1.135		11/16/00 11:13 AM	zeller	New: forwarding of keyboard events from sour...

DDD_3_2_92

New: new resource `bufferGDBOutput' controls whether GDB output is to
be buffered until the next prompt appears, thus making it easier for

Figure 20.4
The CVS Resource History

Resource History View

The *Resource History View* is important for version control because it visualizes the course of versioning. Figure 20.4 shows the Resource History for the file `XXX.YYY` of ASPECTJ. For every revision, it contains the tags, date, author, and comment. By a double-click on the revision number, the revision can be viewed without overriding the working copy. Furthermore, arbitrary revisions can be compared with each other.

20.4 The JAVA editor of ECLIPSE

The JAVA editor of ECLIPSE supports developers in many ways. It

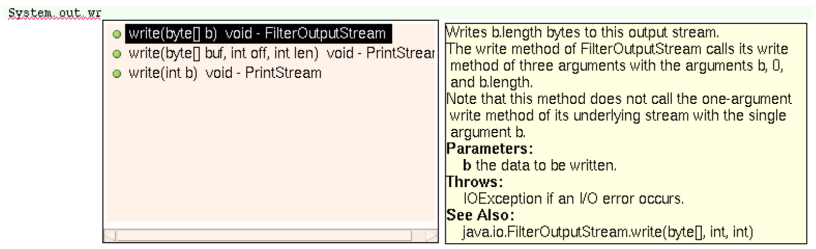
❏ improves presentation and navigation through source code,

- reduces annoying type work,
- checks for errors and *code smells* during editing, and
- present fixes for frequent programming mistakes.

The *syntax highlighting* of the editor improves the readability of source code. If the mouse pointer hovers above an element, additional information (like JAVADOC comments) fades in. ECLIPSE can collapse complete methods or classes to a single line, thus increasing the clarity of large source files. Remarkable are the *quick diffs*, which highlight changes since the last saving or commit to the version archive. The *hyperlink mode* simplifies navigation by making all JAVA elements clickable. For instance, clicking on `Object`, opens the source code of the class `Object`. This mode is enabled, by holding the *Ctrl* key pressed.

Quick diffs

Hyperlink mode



```
System.out.wr
```

- write(byte[] b) void - FilterOutputStream
- write(byte[] buf, int off, int len) void - PrintStrear
- write(int b) void - PrintStream

Writes b.length bytes to this output stream.
The write method of FilterOutputStream calls its write method of three arguments with the arguments b, 0, and b.length.
Note that this method does not call the one-argument write method of its underlying stream with the single argument b.
Parameters:
b the data to be written.
Throws:
IOException if an I/O error occurs.
See Also:
java.io.FilterOutputStream.write(byte[], int, int)

Figure 20.5 *The content assist functionality*

Content assist

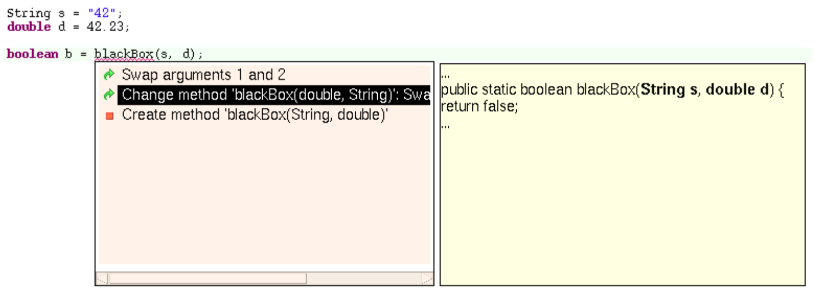
By the use of *content assist* features, developers can reduce annoying type work. For instance, it is sufficient to type the beginning of a class or function name, and then select one of the recommended names (see Figure 20.5). ECLIPSE also takes care of necessary import statements in this scenario. In a similar way, ECLIPSE creates skeletons for JAVADOC comments or loops. For instance, a `/**` and a line break produce a JAVADOC comment, and a `for` makes a `for` loop. Furthermore, ECLIPSE can creates `get` and `set` methods for fields. Content assist features simplify not only development, but also avoid programming mistakes.

Checks for errors and code smells

ECLIPSE checks for errors and *code smells* during programming. In the menu *Windows* → *Preferences* → *Compiler* a multitude of checks can be activated (have a look at the topics *Style*, *Advanced*, *Unused Code*, and *Javadoc*). It is highly recommended to activate most of these checks to recognize potential faults early. Figure 20.2 shows some examples for such warnings in the Problems View. Usually, variables and methods, that are never read or used, are candidates for superfluous code (like in our example `dummy()`). However, in many cases they are indicators for programming errors and undesired effects. The same holds for local variables that hide fields. In our example hides the variable `name` of `main()` the field `name` of class `HelloWorld`.

Thus the field `name` remains `null` and our program print “Hello World, null” in any case. Such errors are avoided by taking warnings seriously.

Additional plug-ins extend the Problems View with more messages. For instance, the CHECKSTYLE plug-in checks for further code smells, including user-defined ones (see also Chapter 17), and displays the results in this view.



```
String s = "42";
double d = 42.23;

boolean b = blackBox(s, d);
```

Swap arguments 1 and 2
Change method 'blackBox(double, String)': Swa
Create method 'blackBox(String, double)'

```
...
public static boolean blackBox(String s, double d) {
return false;
...
```

Figure 20.6
Samples quick fixes

Quick Fix

A special feature of ECLIPSE are the so-called *quick fixes*, which fix an error at the touch of one button. Instances for such quick fixes are: add a type cast, catch an exception, add missing imports, or change the order of parameters in method calls or declarations (see Figure 20.6).

20.5 Program comprehension with JDT

Software engineering separates between *forward* and *reverse engineering*:

Forward Engineering is the *proceeding* development of software: from the specification to the design, and from the design to the finished product.

Reverse Engineering is the process of *identifying* and *displaying* structures and relations in a finished or unfinished product.

Locality

Reverse engineering is not only important during maintenance, but also during earlier phases, for instance, if one has to work in unfamiliar code. The main problem is a *lack of locality*: if you look at an element (e.g., a function, method, class, or a variable) in unfamiliar software, the dependencies and effects of these elements are spread across the entire software. You must therefore try to restore locality by abstraction: important elements are made visible, unimportant elements are not displayed.

For analyzing and understanding large software, command line tools are not well suited because they are text-based. For example, GREP can restore locality, since only the lines containing the text pattern are shown. However, GREP takes no other dependencies into account. In order to detect connections and structures better, we finally have to abstract from the source code, since a textual representation gets confusing even for small programs. *Graphical browsers* use icons and annotations to provide various levels of abstraction, which make intuitive comprehension and navigation easier.

Graphical browser

The JAVA standard library consists of several thousand files and classes—a size that is almost impossible to handle with traditional tools and editors. For instance, if we want to understand what a `Button` in this library is, the *Java Development Tools (JDT)* of ECLIPSE support us in many ways: Besides many special views and perspectives, JDT provides a powerful search engine for JAVA.

Java Development Tools, JDT

20.5.1 The Hierarchy View

Hierarchy View

The *Java Type Hierarchy View* is added to the workbench by *Window* → *Show View* → *Other* → *Java* → *Hierarchy*. The symbol to be shown is selected with *Focus on* from the view, or with *Open Type Hierarchy* of its context menu.

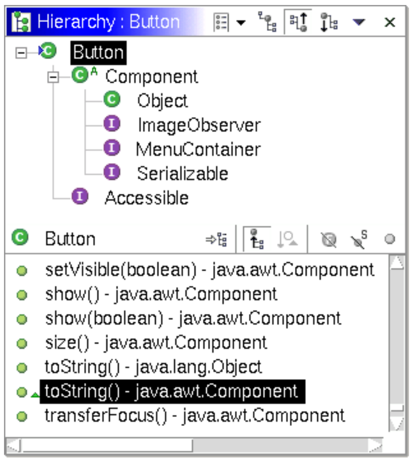

Hierarchy : Button
Button
Component
Object
ImageObserver
MenuContainer
Serializable
Accessible
Button
setVisible(boolean) - java.awt.Component
show() - java.awt.Component
show(boolean) - java.awt.Component
size() - java.awt.Component
toString() - java.lang.Object
toString() - java.awt.Component
transferFocus() - java.awt.Component

Figure 20.7 *The Hierarchy View*

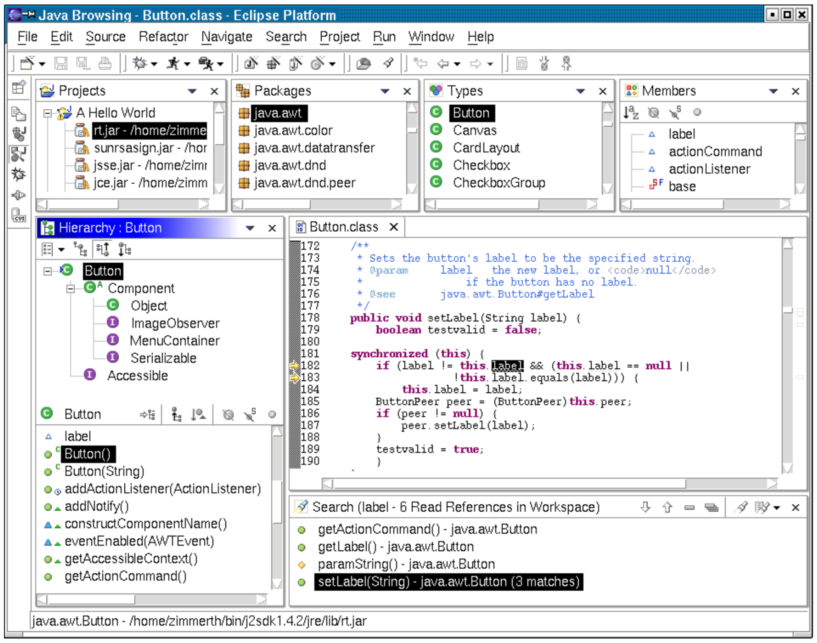
The Hierarchy View in Figure 20.7 shows the elements of the class `Button`. In the upper area of the view we find the inheritance hierarchy of the class `Button`. It contains all classes and interfaces of which `Button` inherits methods or fields. We recognize that `Button` extends the classes `Component` and `Object`. Furthermore, we notice that `Button` implements the interface `Accessible` itself, and for the other interfaces it inherits the implementation of `Component`.

The lower area of the view displays the methods and variable defined by `Button`. With *Show all Inherited Members* we extend this list to all methods and variables that are visible for `Button`. Different icons and colors are used to emphasize the kind, the visibility, and the inheritance of methods. For instance, we recognize that the method `toString()` is inherited of the class `Component`, where the original implementation of the class `Object` was overridden.

20.5.2 The Java Browsing Perspective

Java Browsing Perspective

Another abstraction of the class `Button` is the *Java Browsing Perspective* (see Figure 20.8, *Window* → *Open Perspective* → *Other* → *Java Browsing*). In contrast to the Package Explorer, the projects, packages, types, and members are displayed in separate views, which can be filtered by certain criteria. Thus the developer can switch to other program parts faster. Like every perspective the Java Browsing Perspective can be extended by additional views, e.g., by the Hierarchy View that has been described in the previous section.



```
    /**
     * Sets the button's label to be the specified string.
     * @param     label   the new label, or <code>null</code>
     *                    if the button has no label.
     * @see       java.awt.Button#getLabel
     */
    public void setLabel(String label) {
        boolean testvalid = false;

    synchronized (this) {
        if (label != this.label && (this.label == null ||
                        !this.label.equals(label))) {
            this.label = label;
        ButtonPeer peer = (ButtonPeer)this.peer;
        if (peer != null) {
            peer.setLabel(label);
        }
        testvalid = true;
        }
```

Search (label - 6 Read References in Workspace)

getActionCommand() - java.awt.Button
getLabel() - java.awt.Button
paramString() - java.awt.Button
setLabel(String) - java.awt.Button (3 matches)

java.awt.Button - /home/zimmerth/bin/j2sdk1.4.2/jre/lib/rt.jar

Figure 20.8
Die Java Browsing Perspective

20.5.3 The Call Hierarchy

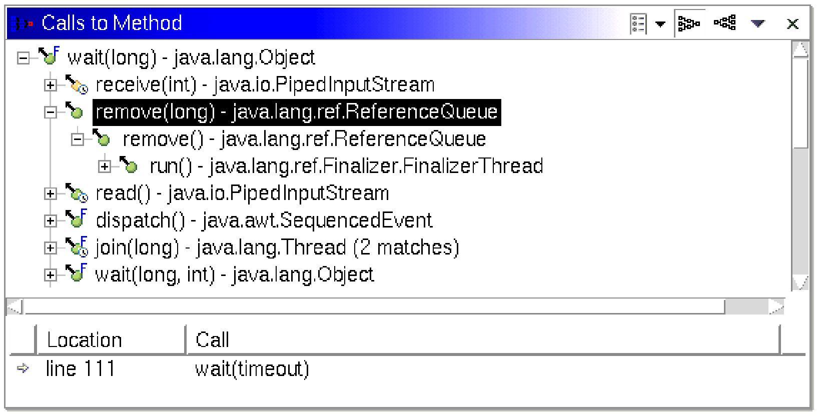
Call Hierarchy

The *Call Hierarchy* is another important abstraction for program comprehension. The Call Hierarchy shows for a given method

❏ of which methods it is called (*caller hierarchy*), and

❏ which methods it calls itself (*callee hierarchy*).

The Call Hierarchy is opened from the context menu of a method by *Open Call Hierarchy*. Figure 20.9 on the following page shows the Caller Hierarchy for the method `wait(long)` of the class `Object`. All methods that call `wait(long)` are displayed in a tree: `wait(long)` is called by the method `remove(long)`, which is called only by `remove()`.



```
Calls to Method
⊟ wait(long) - java.lang.Object
   ⊞ receive(int) - java.io.PipedInputStream
   ⊟ remove(long) - java.lang.ref.ReferenceQueue
      ⊟ remove() - java.lang.ref.ReferenceQueue
         ⊞ run() - java.lang.ref.Finalizer.FinalizerThread
   ⊞ read() - java.io.PipedInputStream
   ⊞ dispatch() - java.awt.SequencedEvent
   ⊞ join(long) - java.lang.Thread (2 matches)
   ⊞ wait(long, int) - java.lang.Object
```

Location	Call
line 111	wait(timeout)

Figure 20.9 *The Call Hierarchy*

20.5.4 Searching in JAVA source code

Syntax-based search

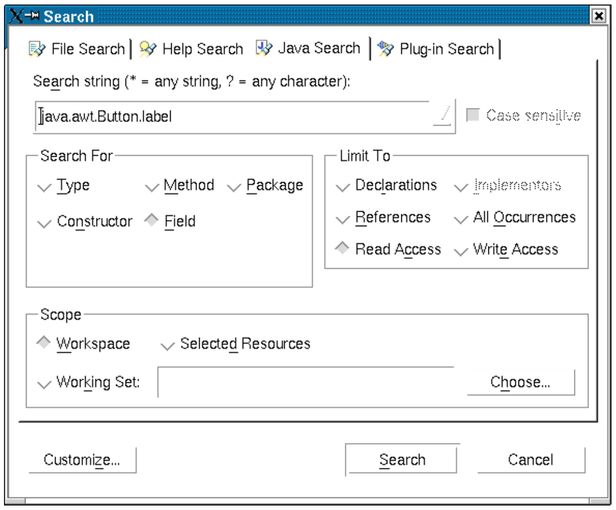
In addition to a traditional *text-based search* (including regular expressions), ECLIPSE allows a *syntax-based search* for JAVA elements (see Figure 20.10 on the next page, *Search* → *Java*). It is possible to search for types, constructors, fields, and packages; the search can be restricted to declarations, references, implementations, read accesses, or write accesses. For instance, this syntax-based search can find all locations that read the variable `label` of `Button`. The results for this query are listed in the *Search View* and marked in the source code with arrows (see Figure 20.8). In our example, there are six read references distributed across four methods: `getActionCommand`, `getLabel`, `paramString`, and `setLabel`. The occurrence of `setLabel` may be surprising, but is reasonable because `setLabel` checks that `label` actually changed, before assigning a new value and notifying its observers.

Search is not limited to single files, if desired, it takes place in the entire workspace or the system libraries. However, for a quick search the command *Occurrences in File* exists in context menus. This command only searches in the currently activated editor for any occurrence of the selection.

Occurrences in File

In contrast to a text-based search, a syntax-based search is more precise and returns no incorrect results. For that reason, syntax-based search is superior to text-based search in most situations. Exceptions are makefiles or text files, where a syntax-based search is unavailable or impossible. In some situations, inaccuracy is desired like for the search in comments.

However, a *semantic-based search* is even more powerful than a syntax-based search. For instance, it finds references across method calls (variables



Search string (* = any string, ? = any character): java.awt.Button.label

Case sensitive

Search For: Type, Method, Package, Constructor, [✓ Field]

Limit To: Declarations, Implementors, References, All Occurrences, [✓ Read Access], Write Access

Scope: [✓ Workspace], Selected Resources, Working Set:

Figure 20.10
The JAVA search in ECLIPSE

are passed as parameters and referenced inside the called method). Such a search needs a *data flow analysis* as described in Chapter 18 and 19.

20.6 Refactoring

Refactoring

A *refactoring* is a semantic preserving code transformation, that improves the structure and quality of programs. Like design patterns there is no general approach, rather a catalog of approaches. Refactorings can be applied in the design, in the implementation, and in the maintenance of software. In practice, refactoring is not always semantic-preserving and therefore some kind of risk. For that reason, refactoring should be performed exclusively with tool support and never manually. Furthermore, a test suite for regression tests is recommended to guarantee semantic-preservation, and a version control to restore previous versions, in case the refactoring fails.

Requirements for refactoring

Refactoring in ECLIPSE

ECLIPSE and JDT support refactoring actively (see menu *Refactor*): on the one hand, ECLIPSE integrates CVS and JUNIT, and on the other hand, JDT provides automatic refactoring procedures. A selection of such procedures is presented below.

Move/Rename classes, methods, and fields. This refactoring adjusts packages, file names, as well as uses of the element to the new name. Even references in comments are handled and for fields the names of `get` and `set` are changed.

Extract Method converts a selection into a new method. The signature is determined automatically, and the selection is replaced by a call to the new method.

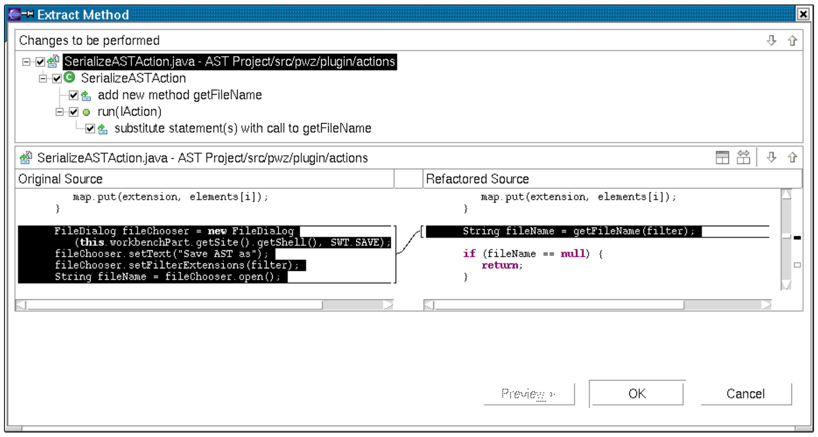
Extract Interface supports the process of creating a new interface from an existing class. The methods that should be part of the new interface are selected in a dialog.

Change Signature modifies the signature of a method. For this, the refactoring adjusts calls to the method and inserts user-specified dummy values for new parameters.

Local Rename is not a pure refactoring because it works only on the opened file and not on the whole project. Local Rename is called like a quick fix and is very helpful to rename local variables of a method.

Organize Imports is not a pure refactoring, either. Organize Imports optimizes import statements: superfluous imports are removed and wildcards like `*` are resolved.

The refactorings of ECLIPSE do not guarantee that the semantic of a program is preserved. Therefore ECLIPSE displays a preview, which lists all changes that will be performed by a refactoring (Figure 20.11). Developers can select desired changes from this list or cancel the complete refactoring.



SerializeASTAction.java - AST Project/src/pwz/plugin/actions
SerializeASTAction
add new method getFileName
run(IAction)
substitute statement(s) with call to getFileName

SerializeASTAction.java - AST Project/src/pwz/plugin/actions

Original Source

```
        map.put(extension, elements[i]);
    }

    FileDialog fileChooser = new FileDialog
        (this.workbenchPart.getSite().getShell(), SWT.SAVE);
    fileChooser.setText("Save AST as");
    fileChooser.setFilterExtensions(filter);
    String fileName = fileChooser.open();
```

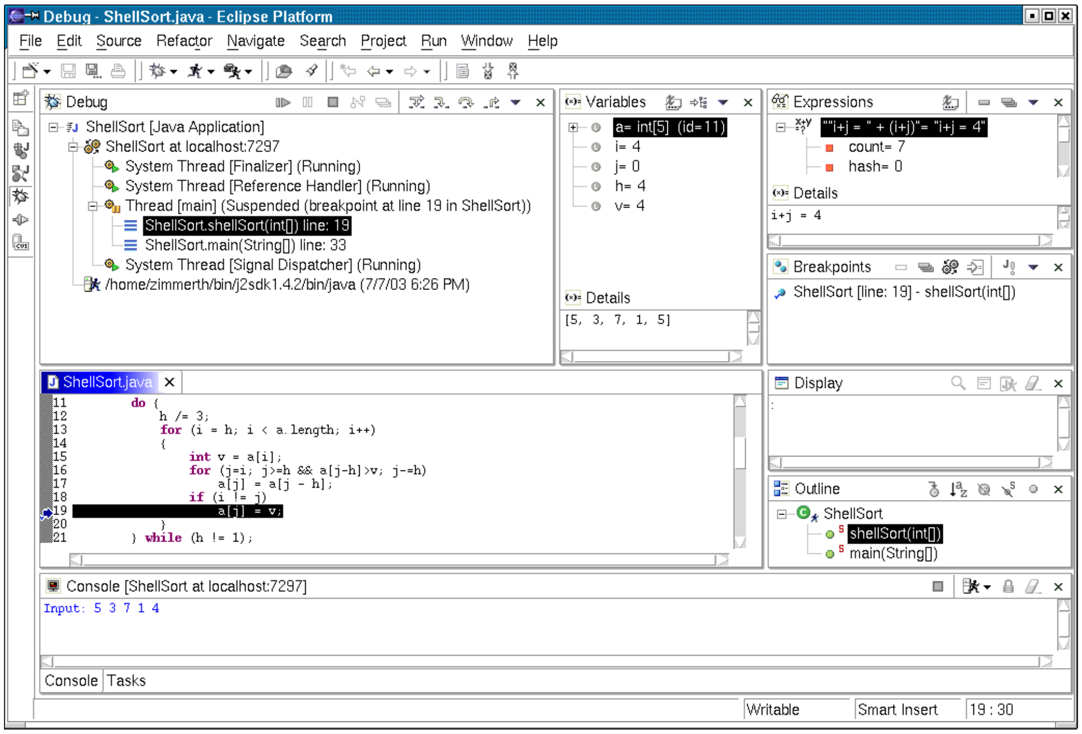
Refactored Source

```
        map.put(extension, elements[i]);
    }

    String fileName = getFileName(filter);

    if (fileName == null) {
        return;
    }
```

Figure 20.11 *The refactoring "Extract Method"*

20.7 Debugging with ECLIPSE

With the JAVA debugger of ECLIPSE we can control the execution of programs, and examine and change program states (for concepts of debugging, see Chapter 15). The debugger supports *hot fix*, that means, modifications on source code affect the running program immediately. Furthermore, it not only supports *local debugging* on the developer's computer, but also *remote debugging* over networks.



```
ShellSort [Java Application]
  ShellSort at localhost:7297
    System Thread [Finalizer] (Running)
    System Thread [Reference Handler] (Running)
    Thread [main] (Suspended (breakpoint at line 19 in ShellSort))
      ShellSort.shellSort(int[]) line: 19
      ShellSort.main(String[]) line: 33
    System Thread [Signal Dispatcher] (Running)
  /home/zimmerth/bin/j2sdk1.4.2/bin/java (7/7/03 6:26 PM)
```

```
a= int[5]  (id=11)
i= 4
j= 0
h= 4
v= 4
[5,  3,  7,  1,  5]
```

```
""i+j = " + (i+j)"= "i+j = 4"
  count= 7
  hash= 0
i+j = 4
```

```
ShellSort [line: 19] - shellSort(int[])
```

```java
        do {
            h /= 3;
            for (i = h; i < a.length; i++)
            {
                int v = a[i];
                for (j=i; j>=h && a[j-h]>v; j-=h)
                    a[j] = a[j - h];
                if (i != j)
                    a[j] = v;
            }
        } while (h != 1);
```

```
ShellSort
  shellSort(int[])
  main(String[])
```

```
Console [ShellSort at localhost:7297]
Input: 5 3 7 1 4
```

Figure 20.12 *The Debug Perspective*

Debug Perspective

For debugging of programs, the developer can access the *Debug Perspective* (see Figure 20.12, *Window* → *Open Perspective* → *Other* → *Debug*), which provides many additional views:

- ❏ The *Debug View* contains processes, threads and call stacks.
- ❏ The *Variables View* shows values of visible variables.
- ❏ The *Breakpoints View* lists all breakpoints.
- ❏ The *Expressions View* evaluates expressions and manages watchpoints.
- ❏ In the *Displays View* we can define displays that are refreshed at every program stop.
- ❏ The *Threads and Monitors View* shows, which threads hold locks, and which threads wait for the release of locks.

20.8 Resource changes and building programs

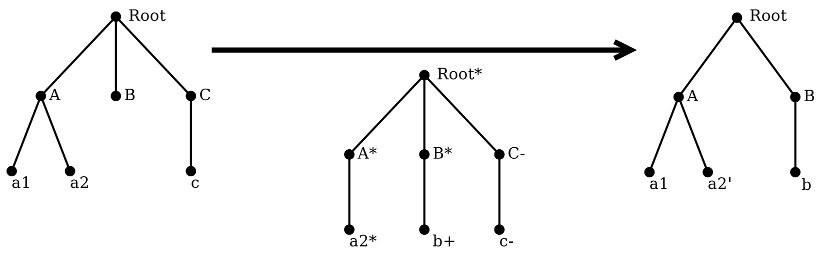
Resource change listener

The different views of ECLIPSE have to be updated if resources change. For this, ECLIPSE uses the concept of *resource change listeners* that are notified at changes.

Resources are organized hierarchically. Thus changes do not affect only the resource itself, but also resources that are in the hierarchy above the changed resource. For instance, ECLIPSE highlights erroneous files and directories with *label decorations*. A directory is erroneous if one of the contained files or subdirectories is erroneous. In case, the developer fixes an erroneous file, ECLIPSE possibly has to refresh the label decorations of all enclosing directories. Therefore ECLIPSE does not just notify resource listeners, but provides them with a *resource delta tree*. A resource delta tree contains in a tree all resources affected by a change. Figure 20.13 shows an example for such a tree: the directory `C`, including all of its files, has been deleted (`-`); the file `b` has been added (`+`) to the directory `B`; and the file `a2` has been modified (`*`). We recognize, that a change on a file always affects the enclosing directories, and that unchanged files, like `a1`, are not contained in a resource delta tree.

Resource delta tree

Figure 20.13
Resource delta tree



Project builder

Resource delta trees are not only important for resource listeners, but also for *project builders*. A project builder gets a resource delta tree and provides several kinds of program construction:

Incremental build

❏ For an *incremental build* (see Section 8.2), the project builder uses the resource delta tree to determine the resources that need to be rebuild. Afterwards it rebuilds only these resources. For instance, a JAVA project builder might recompile only changed JAVA files.

Auto build

❏ A special case of incremental build is *auto build*, which automatically performs an incremental build after every resource change.

Full build

❏ After some changes an incremental build is not possible anymore (e.g., changes on compiler settings), and a *full build* is necessary.

Resource Listener vs. Builder

Both, resource listeners and project builders react to resource changes—what are the differences? A resource listener watches single resources, while a project builder always watches a complete project. Therefore, a resource listener registers at the resource; a project builder, in contrast, registers at ECLIPSE and is assigned to projects that have a specific *project nature*. They also differ in the cost of implementation: Project builders are much more complicated than resource listeners, which are rather light-weight.

Project Natures

20.9 The design of ECLIPSE

ECLIPSE is an open and extensible platform for development tools. This requires a mechanism that allows tool developers to integrate their tools into ECLIPSE without loss of independence. ECLIPSE realizes this idea by its plug-in architecture. At the time of this writing, more than 500 plug-ins exist for ECLIPSE.

20.9.1 The architecture of ECLIPSE

Figure **??** shows the architecture of ECLIPSE. The *Software Development Kit* (SDK) of ECLIPSE consists of three parts:

Eclipse Platform

Eclipse Platform. The *Eclipse Platform* provides the basic functionality of ECLIPSE. It consists of the *Platform Runtime*, which is the core of ECLIPSE and is responsible for loading and managing plug-ins. The Platform Runtime is the only part of ECLIPSE that is not realized as a plug-in; all other components are integrated via the plug-in mechanism. The *Workspace* manages resources (see also Section 20.1) and the *Workbench* realizes the graphical user interface (see also Section 20.2). All other components, like *Team*, *Help* or *Debug*, are built on both, the Workspace and the Workbench.

Java Development Tools, JDT

Java Development Tools (JDT). The *Java Development Tools* extend ECLIPSE in order to provide functionality specific for JAVA, like the views described in Section 20.5.

Plug-in Development Environment, PDE

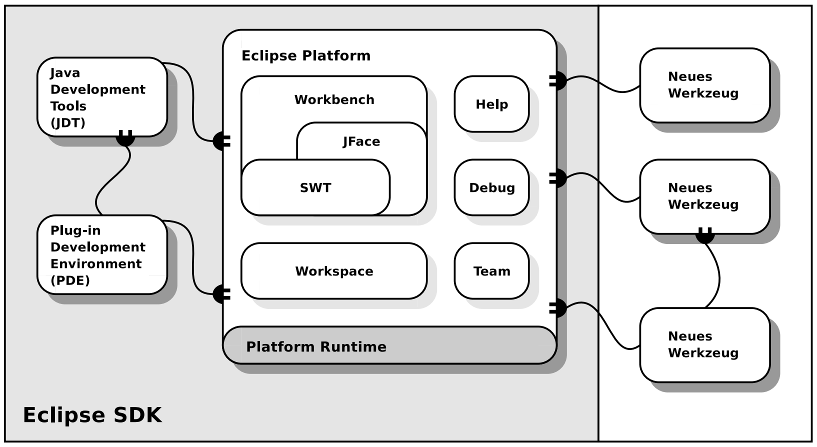
Plug-in Development Environment (PDE). The *Plug-in Development Environment* extends ECLIPSE to provide functionality specific for plug-in development. As plug-ins are developed in JAVA, PDE not only extends the Eclipse Platform, but also JDT.

20.9.2 The plug-in mechanism

Extension point

A *plug-in* is the smallest possible unit that extends ECLIPSE with a new functionality. A plug-in must contribute to at least one *extension point*, which means it provides an implementation for it. In other words, extension points are a mechanism to connect two plug-ins with each other. ECLIPSE defines many extension points itself, but third party plug-ins also can define extension points. A plug-in consists of two parts:

Figure 20.14
The ECLIPSE architecture



Declaration of a plug-in

Declaration. The *declaration* of a plug-in is the *manifest*-file called `plugin.xml`. This XML-file describes which extension points are implemented (at least one), and what new extension points are defined (optional). Furthermore this file contains dependencies to other plug-ins. Example **??** shows a complete manifest-file that is described in detail in Section **??**.

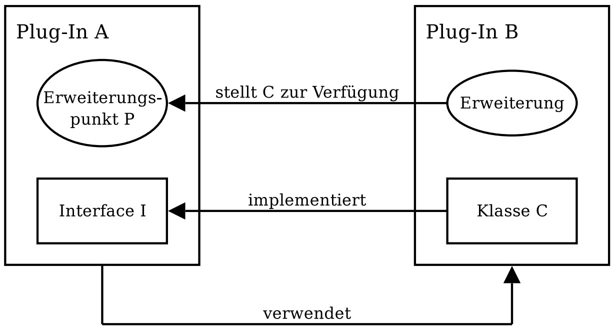
Implementation of a plug-in

Implementation. The *implementation* of a plug-in is written in JAVA. However, some plug-ins need no JAVA implementation; an example is the help of ECLIPSE that is also realized with the plug-in mechanism.

Interaction of plug-ins

Figure **??** illustrates the plug-in concept for two plug-ins. The plug-in `A` defines a new extension point `P` and an interface `I`. In order to use this new extension point, the plug-in `B` has to implement interface `I` with an own class `C`. This class is contributed to the extension point via the manifest-file. Finally, the plug-in `A` can query for extensions, and use located classes, like in our example `C`.

Figure 20.15
The plug-in concept



Simple tools consist of only one plug-in, however, complex tools have multiple plug-ins. Therefore, ECLIPSE allows the combination of several plug-ins into a *feature*. A feature has additional information, like a license and a reference to an update site in the Internet. These updates are installed with the *update manager* of ECLIPSE.

Feature

During the startup of ECLIPSE, the Platform Runtime reads all manifest files and creates the *plug-in registry*. The plug-in registry contains dependencies between plug-ins, and manages contributions to extension points. Plug-ins are not loaded until that moment, where their functionality is called by the user first (*lazy loading*). This approach speeds up the startup of ECLIPSE enormously. The activation of a plug-in may cause the activation of other plug-ins on which the activated plug-in depends.

Plug-in registry

Lazy loading

20.9.3 Integration of tools in ECLIPSE

The complete integration of a tool into ECLIPSE can be very expensive. Therefore several *levels of integration* exist.

Levels of integration in ECLIPSE

No integration. An integration into ECLIPSE is not always necessary, for instance, if the tools has a comfortable user interface and does not exchange data with other tools.

Integration by call. This kind of integration starts the tool in an own process and window. The tools is responsible for the management of resources, but profits of some ECLIPSE components like the version control.

Data integration. The integration of tools is possible with *data sharing*. Tools that follow this approach need an access method, an exchange protocol, and a transformation procedure. ECLIPSE supports data integration by many open standards, like WebDAV, XMI, and XSLT. The risks of data sharing are loss of integrity and the danger of high coupling between tools.

Integration by an API. In this case, a tool provides access to its functionality by an API. However, the user interface of the tool should be separated of its core.

Integration by GUI. This level integrates the tool directly into the ECLIPSE user interface by extending menus, tool bars, views and perspectives.

20.10 A sample plug-in: the ASTView

In this section we present a sample plug-in. The plug-in shows the abstract syntax tree (AST) of a JAVA file in a view (Figure **??**). Internally, ECLIPSE represents JAVA files with such trees. However, they are a cross between a pure abstract syntax tree and a parse tree because ECLIPSE includes brackets and comments.

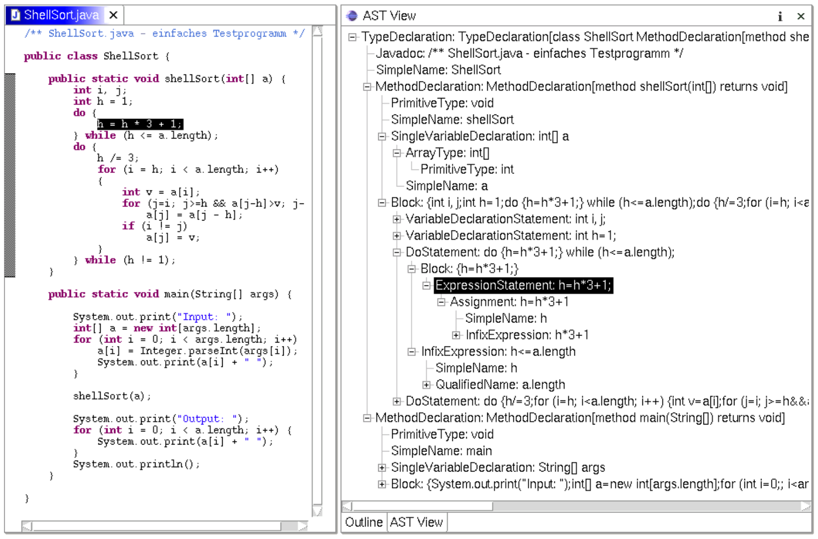


Figure 20.16
The ASTView

```java
/** ShellSort.java - einfaches Testprogramm */

public class ShellSort {

    public static void shellSort(int[] a) {
        int i, j;
        int h = 1;
        do {
            h = h * 3 + 1;
        } while (h <= a.length);
        do {
            h /= 3;
            for (i = h; i < a.length; i++)
            {
                int v = a[i];
                for (j=i; j>=h && a[j-h]>v; j-
                    a[j] = a[j - h];
                if (i != j)
                    a[j] = v;
            }
        } while (h != 1);
    }

    public static void main(String[] args) {

        System.out.print("Input: ");
        int[] a = new int[args.length];
        for (int i = 0; i < args.length; i++)
            a[i] = Integer.parseInt(args[i]);
            System.out.print(a[i] + " ");
        }

        shellSort(a);

        System.out.print("Output: ");
        for (int i = 0; i < a.length; i++) {
            System.out.print(a[i] + " ");
        }
        System.out.println();
    }

}
```

```
⊟ TypeDeclaration: TypeDeclaration[class ShellSort MethodDeclaration[method she
    Javadoc: /** ShellSort.java - einfaches Testprogramm */
    SimpleName: ShellSort
  ⊟ MethodDeclaration: MethodDeclaration[method shellSort(int[]) returns void]
      PrimitiveType: void
      SimpleName: shellSort
    ⊟ SingleVariableDeclaration: int[] a
      ⊟ ArrayType: int[]
          PrimitiveType: int
        SimpleName: a
    ⊟ Block: {int i, j;int h=1;do {h=h*3+1;} while (h<=a.length);do {h/=3;for (i=h; i<a
      ⊞ VariableDeclarationStatement: int i, j;
      ⊞ VariableDeclarationStatement: int h=1;
      ⊟ DoStatement: do {h=h*3+1;} while (h<=a.length);
        ⊟ Block: {h=h*3+1;}
          ⊟ ExpressionStatement: h=h*3+1;
            ⊟ Assignment: h=h*3+1
                SimpleName: h
              ⊞ InfixExpression: h*3+1
        ⊟ InfixExpression: h<=a.length
            SimpleName: h
          ⊞ QualifiedName: a.length
      ⊞ DoStatement: do {h/=3;for (i=h; i<a.length; i++) {int v=a[i];for (j=i; j>=h&&
  ⊟ MethodDeclaration: MethodDeclaration[method main(String[]) returns void]
      PrimitiveType: void
      SimpleName: main
    ⊞ SingleVariableDeclaration: String[] args
    ⊞ Block: {System.out.print("Input: ");int[] a=new int[args.length];for (int i=0;; i<ar
```

The plug-in provides the following functionality:

- ❏ Select a JAVA file from its context menu.
- ❏ Show the AST for this file in a view.
- ❏ Open vertices in an editor at a double-click.

We now explain snippets of the plug-in. The complete source code is printed on page **??**–**??**. For trying the plug-in yourself, please consider the notes in the practice section on page **??**.

Declare the Plug-in

The file `plugin.xml` (see Example **??**) contains the declaration of a plug-in. Besides general information and a reference to the code archive (`ASTView.jar`), the file contains dependencies to other plug-ins:

- ❏ `org.eclipse.core.resources` for access to resources,
- ❏ `org.eclipse.ui` for the user interface,
- ❏ `org.eclipse.jdt.core` for JAVA specific functionality, and
- ❏ `org.eclipse.jdt.ui` for the JAVA editor.

In order to extend ECLIPSE with a new view, we use the extension point `org.eclipse.ui.views`:

```
<extension
      point="org.eclipse.ui.views">
   <category
         name="Wiley"
         id="astview.category.wiley">
   </category>
   <view
         name="AST View"
         icon="icons/sample.gif"
         category="astview.category.wiley"
         class="astview.views.ASTView"
         id="astview.views.ASTView">
   </view>
</extension>
```

By the use of `category` we extend the dialog *Windows → Show View → Other* with a new category named "Wiley" (`name`). Using `view`, we insert a new view in this category. The view is called "AST View" and realized in the class "astview.views.ASTView" (`class`). Our view is identified throughout ECLIPSE by the value in `id`.

Furthermore we want to extend all context menus for JAVA files. The respective extension point is `org.eclipse.ui.popupMenus`:

```
<extension
      point="org.eclipse.ui.popupMenus">
   <objectContribution
         objectClass=
            "org.eclipse.jdt.core.ICompilationUnit"
         id="astview.ICompilationUnit">
      <action
            label="Update AST View"
            class="astview.actions.UpdateViewAction"
            enablesFor="1"
            id="astview.actions.UpdateView">
      </action>
   </objectContribution>
</extension>
```

To show our entry only for JAVA files, we define an `objectContribution` that shows entries only for elements of type `objectClass`. We select `ICompilationUnit` as type because this class represents JAVA files in ECLIPSE. The action will be called "Update AST View" (`label`), and is enabled for exactly one selected element (`enablesFor`). In case the user selects our action, ECLIPSE calls the class "astview.actions.UpdateViewAction" (`class`).

Create the AST

In ECLIPSE, the creation of an AST from a JAVA file is simple:

```
ICompilationUnit unit = ...;
ASTParser parser = ASTParser.newParser(AST.JLS2);
parser.setSource(unit);
CompilationUnit root =
    (CompilationUnit) parser.createAST(null);
```

The variable `unit` represents the JAVA file of interest and is determined by selection of the context menu (see below).

Display the AST

TreeViewer

We display the AST in a view by the use of a *TreeViewer*. A TreeViewer consists of four parts:

Input. The *input data* will be displayed by the TreeViewer. For our plug-in the input data is the root of the AST, because it can be used to access the complete AST.

ContentProvider. A *ContentProvider* maps the input to the structure of the viewer. In our example the structure is a tree, and we have to implement methods to access the parents (`getParent()`) and children of a node (`getChildren()` and `hasChildren()`). The parent is stored in nodes, but for determining the children we have to traverse the AST with an implementation of an *ASTVisitor*.

LabelProvider. A *LabelProvider* defines the format of tree elements. The text is determined by `getText()` and an optional icon by `getImage()`. For our example, we simply use the class name of a node and its `toString`-representation.

Sorter. A *Sorter* sorts the displayed input. For our example we need no Sorter.

Update the view

Whenever the user selects the action *Update AST View* in a context menu, the method `run()` of the class `UpdateViewAction` is called. This method performs the following steps:

❏ Determine the selected JAVA file: We take the first and only element of the selection (`selection`) of the context menu.

```
Object obj = ((IStructuredSelection)
   selection).getFirstElement();
ICompilationUnit unit = (ICompilationUnit) obj;
```

- ❏ Find the ASTView: Before updating the view, we have to find it. For that purpose, we use the workbench and the identifier of the view, to determine the `IViewPart` that represents our ASTView.

```
IWorkbenchPage page = workbenchPart.getSite().
   getWorkbenchWindow().getActivePage();
IViewPart vp =
   page.findView("astview.views.ASTView");
```

Open Elements in an Editor

In order to open elements in an editor, we define an action called `doubleClickAction`. The `run()`-method of this action determines the element, on which the user double-clicked (`node`), and opens this element in an editor:

```
IJavaElement elem =
   unit.getElementAt(node.getStartPosition());
IEditorPart javaEditor = JavaUI.openInEditor(elem);
JavaUI.revealInEditor(javaEditor, elem);
```

The action has to be added to the TreeViewer with a call to the method `addDoubleClickListener()`.

20.11 Automation and intuition

In the previous sections, we showed how the requirements for integrated environments, such as *intuitive operation*, *interoperability*, *abstraction*, and *functionality* are met by ECLIPSE. Compared to traditional command line tools, ECLIPSE has clear advantages in the area of abstraction and intuitive operation.

But what about the last requirement, that of *automation*? Command line tools can be automated without any major problems by the use of shell scripts. Usually, integrated environments have large deficits here, because an operation is only possible via the graphical user interface; unfortunately, this holds for ECLIPSE as well. However, most tools integrated by ECLIPSE, like JUNIT, ANT, and CVS, are available as command line tools as well. Therefore the lack of automation does not matter very much.

For ECLIPSE, a more important issue, is the growing number of plug-ins. Every day, more plug-ins are published that are not mature enough. Furthermore, the support for other programming languages lags behind the support of JAVA. However, the ECLIPSE boom has just begun, and we do not know its future yet.

Concepts

- ❏ *Integrated environments* provide various tools in a shared, unified, and intuitively operational environment.
- ❏ *Open environments* allow the integration of external tools. *Access interfaces* to the data and functions of the environment are useful (like the ECLIPSE API).
- ❏ *High integration* provides programmers with a uniform user interface, *modularity* allows the individual further development of external tools, and *interoperability* enables the cooperation and exchange between integrated tools.
- ❏ An user interface divides into *editors*, *views*, and *perspectives*. *Graphical browsers* provide different views with various levels of abstraction.
- ❏ *Refactorings* improve the structure of programs.
- ❏ *Incremental builder* enable editing of source code *on-the-fly* by efficient compiling techniques.
- ❏ *Plug-ins* allow the flexible extension of ECLIPSE.
- ❏ *Extension points* are potential locations for extensions. All contributed extensions are managed in the *plug-in registry*.

Exercises

1. Discuss the advantages and the potential risks of an open development environment, like ECLIPSE. Why develop many company commercial plug-ins for ECLIPSE, anyway?
2. We have introduced different views on JAVA source code. What views are reasonable for other programming languages, like C, C++, PYTHON, PERL und PHP?
3. Not all tools presented in this book are integrated in ECLIPSE. Is an integration of the following tools reasonable, and if yes how can it look like?
   (a) AUTOCONF (Chapter 9)
   (b) DEJAGNU (Chapter 12)
   (c) BUGZILLA (Chapter 14)
   (d) GPROF and GCOV (Chapter 16)
   (e) CHECKSTYLE (Chapter 17)

   (f) LINT (Chapter 18)
   (g) UNRAVEL (Chapter 19)

   If an integration is not reasonable, is it possible to integrate some of the tool's concepts?

4. Why uses ECLIPSE its own widget toolkit (JFACE and SWT), and not the default JAVA toolkits (AWT and SWING)? Research on the Internet.
5. Your company develops an editor that is based on AWT and SWING. Your boss likes ECLIPSE and wants to integrate it into ECLIPSE. Compare the costs and benefits of the different kinds of integration to each other, and plan the integration. Research on the Internet, whether a direct integration of AWT and SWING components is possible.
6. We only considered integration from the user's and developer's perspective. What other perspective can you think of?
7. There exist more than 500 plug-ins. What are the disadvantages of such a huge number? Figure out a reasonable categorization for plug-ins. Are there plug-in catalogs on the Internet?
8. Research on the ECLIPSE *Rich Client Platform*. Make up the future of ECLIPSE on the *dark side of the moon*. What about a *Sun*?

Bibliographic notes

The first integrated development environment was SMALLTALK, where the graphical programming environment was part of the system and was described in **?**. SMALLTALK was developed together with graphical workstations.

Earlier software development environments were strongly *syntax oriented*: The source code was input for a *syntax controlled* editor, and the program always was kept in a compiled state. These include: PSG from **?** and the *Synthesizer Generator* from **?**.

The term *source code engineering*, coined by **?**, describes the activities taken on by many development environments. In the meantime, the research has broken loose from its source code orientation and now looks at *software engineering environments*, that are oriented to the entire software development process.

? describes the architecture of ECLIPSE. The operation of ECLIPSE is topic of **?**, as well as of **?**. In contrast, **?** focus on the development of plug-ins. You find recent information on ECLIPSE at:

`http://www.eclipse.org/`

Other programming environments

Most commercial compilers come in an *integrated development environment* (*IDE*). However, apart from integrated GUI builders, they rarely support more than the *edit-compile-debug cycle*. Typical programming environments, such as CODEWARRIOR, NETBEANS, the *Visual* series of *Microsoft*, or the *VisualAge* series of IBM, have the problem that many visualization tools are only available for sources that have been compiled *without any errors*.

There are only a few cross-platform environments that support several programming languages, version control systems, and external tools:

❏ SNiFF+ by *Wind River* is an open environment that supports several programming languages and operating systems. External tools are integrated by *adapters*. SNiFF+ provides graphical browsers that are similar to those of ECLIPSE. However, the focus of SNiFF+ is on code analysis

❏ SOURCE NAVIGATOR is very similar to SNiFF+, and also focuses on code analysis. It supports the integration of external tools, and is the only *open* environment, besides ECLIPSE and SNiFF+.

❏ C-FORGE by *Code Forge* supports more than 30 programming languages and several version control systems. Like ECLIPSE, it provides searches and browsers for source code, as well as debugging tools.

❏ CODEBEAMER by *Intland* is a web-based development environment, that supports program comprehension and communication between developers. A web-interface accesses version control systems, and examines source code with several tools. CODEBEAMER is a enhancement for existing development environments; consequently, there is a CODEBEAMER plug-in for ECLIPSE.

❏ TOGETHER by *Borland* contains an integrated UML editor and provides many skeletons for design patterns. Another highlight are the numerous software metrics. TOGETHER also exists as a plug-in for ECLIPSE.

Sometimes, ECLIPSE is called the “EMACS of the 21. century”. The EMACS editor, or the “grandmother” of all editors, is famous since ever. It can be extended by LISP , and many external programming tools are supported. This even goes to real programming environments, like JDE for JAVA. Besides programming, with EMACS one can write emails, surf in the Internet—or write books like this one . . .

Exercices VII

In this exercise you create the *Rational* example of Chapter 13 in ECLIPSE. Furthermore you develop your own plug-in: the *ASTView* presented in Chapter 20.

Introduction to ECLIPSE

1. Download and install ECLIPSE.
2. Create a new JAVA project named `Rational` (use *File* → *New* → *Project*).
3. Create a class `Rational` that contains methods for comparison (`equals`, `compareTo`), and for the four basic arithmetics (`add`, `subtract`, `multiply`, `divide`).
4. Create a tester named `RationalTest` as described in Chapter 13 (for methods `equals`, `compareTo`). Use the dialog *File* → *New* → *Other* → *Java* → *JUnit* → *Test Case*.
5. Execute `RationalTest` with *Run* → *Run As* → *JUnit Test*, and check whether `Rational` passes all test cases.
6. Now create a `RationalArithTest` class, that contains test cases for the basic arithmetics (`add`, `subtract`, `multiply`, `divide`).
7. Create a test suite named `AllRationalTests`, that combines the tests of `RationalTest` and `RationalArithTest`. Use the dialog *File* → *New* → *Other* → *Java* → *JUnit* → *Test Suite*. Check with this test suite, whether your `Rational` class passes all tests.
8. Compare the current version of the class `Rational` with a previous version from the local history (in the context menu *Compare with* → *Local History*).
9. Reconstruct the version history of an arbitrary open-source project with the tools of ECLIPSE. (You will find some open-source projects on `www.sourceforge.net`).

10. Explore the JAVA standard library (`rt.jar`) with ECLIPSE. Start with the *Java Browsing* Perspective or with the *hyperlink mode* (press key *Ctrl*). Possibly, you have to add the JAVA source code as a *source attachment*. You will find the source in the archive `src.zip` in the main directory of your JAVA installation.

Plug-in Development in ECLIPSE

1. Create a *Hello World* plug-in before trying out the *ASTView* plug-in. You will find instructions on the Internet, e.g., in the article “Your First Plug-in” on `www.eclipse.org`.

2. Get familiar with the ECLIPSE help (*Help* → *Help Contents*). Use it to research on:

   ❏ abstract syntax trees (AST)

   ❏ actions

   ❏ viewers, particularly, the TreeViewer

   ❏ the extension point `org.eclipse.ui.views`

   ❏ the extension point `org.eclipse.ui.popupMenus`

3. Create a new plug-in project named `ASTView`. In the *New File* dialog, use the wizard *Plug-in Project* and name the class that controls the plug-in life-cycle `astview.ASTViewPlugin` (case-sensitive). ECLIPSE should automatically import the correct libraries.

4. Open the *manifest editor* for the file `plug-in.xml`, and create the plug-in description (see Example ??). You can enter the source code in the page *plugin.xml* or use the input fields on the other pages of the editor.

5. Call the command *PDE Tools* → *Update Classpath* in the context menu of `plugin.xml`, to refresh the list of libraries that will be imported by ECLIPSE.

6. Now you can test the declaration of our *ASTView*: Start the plug-in test environment with *Run* → *Run As* → *Runtime Workbench*. A second workbench opens; check whether the dialog *Windows* → *Show View* → *Other* contains the entry “Wiley”. Close the test workbench afterwards.

7. Use the Resource Perspective to create a directory called `icons` and insert an arbitrary image named `sample.gif`. This image will be the logo of the *ASTView*.

8. In order to finish the plug-in, implement the remaining classes `ASTView` and `UpdateViewAction` (see Examples **??** and **??**). If you are unfamiliar with an operation, look it up in the ECLIPSE help.

Example 20.1
Plug-in description
`plugin.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id="ASTView"
   name="ASTView Plug-in"
   version="1.0.0"
   provider-name=""
   class="astview.ASTViewPlugin">

   <runtime>
      <library name="astview.jar"/>
   </runtime>
   <requires>
      <import plugin="org.eclipse.core.runtime.compatibility"/>
      <import plugin="org.eclipse.ui.ide"/>
      <import plugin="org.eclipse.ui.views"/>
      <import plugin="org.eclipse.jface.text"/>
      <import plugin="org.eclipse.ui.workbench.texteditor"/>
      <import plugin="org.eclipse.ui.editors"/>
      <import plugin="org.eclipse.core.resources"/>
      <import plugin="org.eclipse.ui"/>
      <import plugin="org.eclipse.jdt.core"/>
      <import plugin="org.eclipse.jdt.ui"/>
   </requires>

   <extension
         point="org.eclipse.ui.views">
      <category
            name="Wiley"
            id="astview.category.wiley">
      </category>
      <view
            name="AST View"
            icon="icons/sample.gif"
            category="astview.category.wiley"
            class="astview.views.ASTView"
            id="astview.views.ASTView">
      </view>
   </extension>

   <extension
         point="org.eclipse.ui.perspectiveExtensions">
      <perspectiveExtension
            targetID="org.eclipse.jdt.ui.JavaPerspective">
         <view
               relative="org.eclipse.ui.views.ContentOutline"
               relationship="stack"
               id="astview.views.ASTView">
         </view>
      </perspectiveExtension>
   </extension>

   <extension
         point="org.eclipse.ui.popupMenus">
      <objectContribution
            objectClass="org.eclipse.jdt.core.ICompilationUnit"
            id="astview.ICompilationUnit">
         <action
               label="Update AST View"
               class="astview.actions.UpdateViewAction"
               enablesFor="1"
               id="astview.actions.UpdateView">
         </action>
      </objectContribution>
   </extension>


</plugin>
```

Example 20.2
Class `ASTView`

```
package astview.views;

import java.util.ArrayList;

import org.eclipse.core.resources.ResourcesPlugin;
import org.eclipse.jdt.core.ICompilationUnit;
import org.eclipse.jdt.core.IJavaElement;
import org.eclipse.jdt.core.JavaModelException;
import org.eclipse.jdt.core.dom.AST;
import org.eclipse.jdt.core.dom.ASTNode;
import org.eclipse.jdt.core.dom.ASTParser;
import org.eclipse.jdt.core.dom.ASTVisitor;
import org.eclipse.jdt.core.dom.CompilationUnit;
import org.eclipse.jdt.ui.JavaUI;
import org.eclipse.jface.action.Action;
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.jface.viewers.DoubleClickEvent;
import org.eclipse.jface.viewers.IDoubleClickListener;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.jface.viewers.ITreeContentProvider;
import org.eclipse.jface.viewers.LabelProvider;
import org.eclipse.jface.viewers.TreeViewer;
import org.eclipse.jface.viewers.Viewer;
import org.eclipse.swt.SWT;
import org.eclipse.swt.graphics.Image;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Shell;
import org.eclipse.ui.IActionBars;
import org.eclipse.ui.IEditorPart;
import org.eclipse.ui.ISharedImages;
import org.eclipse.ui.PartInitException;
import org.eclipse.ui.PlatformUI;
import org.eclipse.ui.part.ViewPart;


/**
 * Class OurVisitor
 */
class OurVisitor extends ASTVisitor {

   private int currentDepth;
   private int maxDepth;
   private ArrayList children;

   public OurVisitor(int depth) {
      this.currentDepth = 0;
      this.maxDepth = depth;
      children = new ArrayList();
   }

   public void postVisit(ASTNode node) {
      currentDepth--;
   }

   public void preVisit(ASTNode node) {
      if (currentDepth > 0 && currentDepth <= maxDepth) {
         children.add(node);
      }
      currentDepth++;
   }

   public Object[] getChildren() {
      return children.toArray();
   }
}


/**
 * Class ASTView
 */
public class ASTView extends ViewPart {

   private TreeViewer viewer;
   private Action sampleAction;
   private Action doubleClickAction;
   private ICompilationUnit unit;

   /**
    * Inner Class ViewContentProvider
    */
   class ViewContentProvider implements ITreeContentProvider {

      public void inputChanged
         (Viewer v, Object oldInput, Object newInput) {
      }

      public void dispose() {
      }

      public Object[] getElements(Object parent) {
         return getChildren(parent);
      }

      public Object getParent(Object child) {
         if (child instanceof ASTNode) {
            return ((ASTNode)child).getParent();
         }
         return null;
      }

      public Object [] getChildren(Object parent) {
         if (parent instanceof ASTNode) {
            OurVisitor myVisitor = new OurVisitor(1);
            ((ASTNode) parent).accept(myVisitor);
            return myVisitor.getChildren();
         }
         return new Object[0];
      }

      public boolean hasChildren(Object parent) {
         if (parent instanceof ASTNode) {
            OurVisitor myVisitor = new OurVisitor(1);
            ((ASTNode) parent).accept(myVisitor);
            return myVisitor.getChildren().length>0;
         }
         return false;
      }
   }


   /**
    * Inner Class ViewLabelProvider
    */
   class ViewLabelProvider extends LabelProvider {

      public String getText(Object obj) {
         String[] className =
            obj.getClass().getName().split("\\.");
         return className[className.length - 1]
            + ": " + obj.toString();
      }
      public Image getImage(Object obj) {
         return null;
      }
   }


   /**
    * Source code of ASTView
    */
   public ASTView() {}


   public void createPartControl(Composite parent) {
      viewer = new TreeViewer
         (parent, SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);
      viewer.setContentProvider(new ViewContentProvider());
      viewer.setLabelProvider(new ViewLabelProvider());
      viewer.setSorter(null);
      viewer.setInput(ResourcesPlugin.getWorkspace());

      makeActions();
      hookDoubleClickAction();
      contributeToActionBars();
   }


   private void makeActions() {

      sampleAction = new Action() {
         public void run() {
            Shell shell = Display.getDefault().getActiveShell();
            MessageDialog.openInformation
               (shell, "Sample Action", "A message dialog.");
         }
      };
      sampleAction.setImageDescriptor
         (PlatformUI.getWorkbench().getSharedImages().
         getImageDescriptor(ISharedImages.IMG_OBJS_INFO_TSK));

      doubleClickAction = new Action() {
         public void run() {
            IStructuredSelection selection =
               (IStructuredSelection) viewer.getSelection();
            ASTNode node = (ASTNode) selection.getFirstElement();
            try {
               IJavaElement elem =
                  unit.getElementAt(node.getStartPosition());
               IEditorPart javaEditor = JavaUI.openInEditor(elem);
               JavaUI.revealInEditor(javaEditor, elem);
            } catch (JavaModelException e) {
               e.printStackTrace();
            } catch (PartInitException e) {
               e.printStackTrace();
            }
         }
      };
   }


   private void hookDoubleClickAction() {
      viewer.addDoubleClickListener(new IDoubleClickListener() {
         public void doubleClick(DoubleClickEvent event) {
            doubleClickAction.run();
         }
      });
   }


   private void contributeToActionBars() {
      IActionBars bars = getViewSite().getActionBars();
      bars.getToolBarManager().add(sampleAction);
   }


   public void setFocus() {
      viewer.getControl().setFocus();
   }


   public void setInput(ICompilationUnit unit) {
      this.unit = unit;
      ASTParser c = ASTParser.newParser(AST.JLS2);
      c.setSource(unit);
      c.setResolveBindings(false);
      CompilationUnit root =
         (CompilationUnit) c.createAST(null);
      viewer.setInput(root);
   }
}
```

Example 20.3
Class
`UpdateViewAction`

```
package astview.actions;

import org.eclipse.jdt.core.ICompilationUnit;
import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.IStructuredSelection;
import org.eclipse.ui.IObjectActionDelegate;
import org.eclipse.ui.IViewPart;
import org.eclipse.ui.IWorkbenchPage;
import org.eclipse.ui.IWorkbenchPart;

import astview.views.ASTView;


public class UpdateViewAction implements IObjectActionDelegate {

   private IWorkbenchPart workbenchPart;
   private ISelection selection;

   public void setActivePart(IAction action,
                             IWorkbenchPart targetPart)
   {
      workbenchPart = targetPart;
   }

   public void run(IAction action) {
      Object obj =
         ((IStructuredSelection) selection).getFirstElement();
      ICompilationUnit unit = (ICompilationUnit) obj;

      IWorkbenchPage page =
          workbenchPart.getSite().getWorkbenchWindow().
          getActivePage();

      IViewPart vp = page.findView("astview.views.ASTView");

      if (vp instanceof ASTView) {
         ((ASTView) vp).setInput(unit);
      }

   }

   public void selectionChanged(IAction action,
                                ISelection selection)
   {
      this.selection = selection;
   }
}
```